**PE-taak: performance onderzoek**



Groepsleden:

* Emre Demirsatan
* Muhammed Genç

Inhoud

[1. Inleiding 3](#_Toc498032075)

[1.1 Wat is performance onderzoek ? 3](#_Toc498032076)

[1.2 Doel 3](#_Toc498032077)

[1.3 Referentie project 3](#_Toc498032078)

[2. Performance Analysis 4](#_Toc498032079)

[2.1 Metrieken 4](#_Toc498032080)

[2.2 Tools 4](#_Toc498032081)

[3. Oplossingsmethode Game Of Life 5](#_Toc498032082)

[3.1 Referentie Algoritme 5](#_Toc498032083)

[3.2 Performance Optimalisatie 6](#_Toc498032084)

[3.2.1 Memory 6](#_Toc498032085)

[3.2.2 Heap 7](#_Toc498032086)

[3.2.3 Final Keyword 8](#_Toc498032087)

[3.2.4 Multithreading 9](#_Toc498032088)

[4. Persoonlijke Reflectie 10](#_Toc498032089)

[4.1.1 Emre Demirsatan 10](#_Toc498032090)

[4.1.2 Muhammed Genc 10](#_Toc498032091)

[5. Bronvermelding 11](#_Toc498032092)

# Inleiding

## Wat is performance onderzoek ?

De letterlijke betekenis van *performance*: ‘het vermogen om veel en/of goed te presteren, prestatie’.

Een performance onderzoek is dus een onderzoek naar het prestatie van het programma. Aan de hand van een performance onderzoek weten de ontwikkelaars wat veel tijd in beslag neemt en/of er nog eventueel wijzigingen kunnen doorgevoerd worden om het programma sneller, beter te laten werken.

Immers is het zo dat een performance onderzoek een programma ontleed en gedetailleerd informatie weergeeft over het geheugengebruik, uitvoeringstijd,...

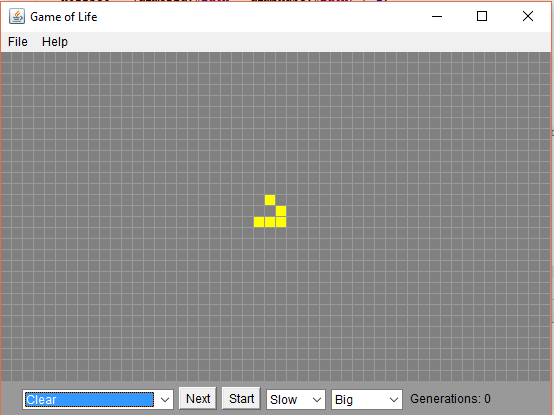
## Doel

Het doel van dit document is het analyseren en verbeteren van source code van Game Of Life. Hierbij gebruiken we een set van leerrijke tools aangeleerd tijdens de les maar ook een deel van tools die zelfstudie van ons vereist.

Dit document zal gedetailleerd de performance oplossing omschrijven die wij toepassen op het project van Game of Life. Grafieken zullen de prestaties vergelijken van onze verbeteringen ten opzichte van de originele code. Wij hebben gekozen om Performance analysis toe te passen in de programmeertaal Java.

## Referentie project

Als referentie hebben we gekozen voor de site van Game Of Life. De project die terug te vinden is op deze link: https://bitstorm.org/gameoflife/. De code bevat gedetailleerd omschreven commentaar waardoor het analyseren van de code een stuk gemakkelijker gaat verlopen.



# Performance Analysis

## Metrieken

Als meting voor het bepalen dat een stuk code verbeterd is maken we gebruik van enkele eigenschappen.

* Heap Size
* Snelheid (uitgedrukt in milliseconden)
* CPU cores utilisatie

De verbeterde snelheid wordt weergegeven als een percentage omdat de snelheid voor een groot deel afhankelijk is van de prestatie van uw processor. Hierbij wordt ook aangeduid wat de originele source code was en wat wij gewijzigd hebben om de prestatie te verbeteren.

## Tools

**JConsole**

Komt standaard meegeleverd met de JDK, installatie is niet vereist. JConsole is uitgerust met een dashboard die doormiddel van grafieken de volgende eigenschappen meegeeft:

* Een weergaven over de total heap size ruimte dat er in beslag wordt genomen in het RAM-geheugen.
* Toont het aantal threads die het programma in beslag neemt.
* Een overzicht over hoeveel procent van de capaciteit van de in processor in beslag wordt genomen.

**YourKit**

YourKit biedt ons de mogelijk om te kijken hoe lang het duurt tot de functie gedaan heeft met uitvoeren. Dit is handig om te kijken precies in welke functie versnellingen kunnen worden toegepast. We zullen gebruik maken van de trial version van YourKit samen met IntelliJ.

**JProfiler**

JProfiler toont meer detail over de garbage collection die moet gebeuren. Hierbij toont JProfiler zelfs wat voor soort objecten zich in de Heap bevinden. Dit kan handig zijn om referenties naar een klasse te leggen in plaats van een nieuwe instantie aan te maken. Door middel van het nemen van een snapshot is het ook mogelijk om een algemeen overzicht te krijgen van de Heap.

# Oplossingsmethode Game Of Life

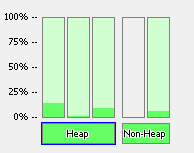
## Referentie Algoritme

Nadat een aantal cellen van een raster is ingekleurd, begint het spel. Game of Life werkt met 'generaties'; om te bepalen of een cel gekleurd ('levend') is of juist niet ('dood') in de volgende generatie wordt er een aantal regels toegepast aan de hand van de status (levend of dood) die de buurcellen hebben. Elke cel heeft 8 buurcellen.

* Als een cel door 2 of 3 gekleurde buurcellen omgeven wordt, blijft deze cel zelf ook gekleurd, zoals in het voorbeeld hieronder.  
  [Blijvenleven.PNG](https://commons.wikimedia.org/wiki/File:Blijvenleven.PNG)  
  In deze afbeelding blijft de middelste cel gekleurd, want de cel wordt omgeven door 2 andere gekleurde cellen.
* Als een cel door 4 of meer gekleurde buurcellen omgeven wordt, gaat deze cel dood door 'overbevolking' (dat wil zeggen, de cel wordt wit). Als een cel door minder dan twee gekleurde buurcellen omgeven wordt, gaat deze cel ook dood, maar dan door eenzaamheid. Zie voorbeeld hieronder.  
  [Doodgaan.PNG](https://commons.wikimedia.org/wiki/File:Doodgaan.PNG)  
  In deze afbeelding gaat de middelste cel dood, want de cel wordt door meer dan 3 of minder dan 2 gekleurde cellen omgeven.
* Als een dode cel wordt omgeven door precies 3 gekleurde buurcellen, wordt deze dode cel ook gekleurd ('geboren'), zoals in het voorbeeld hieronder.  
  [Geborenworden.PNG](https://commons.wikimedia.org/wiki/File:Geborenworden.PNG)  
  In deze afbeelding wordt de middelste cel geboren, want de cel wordt door exact 3 gekleurde cellen omgeven.

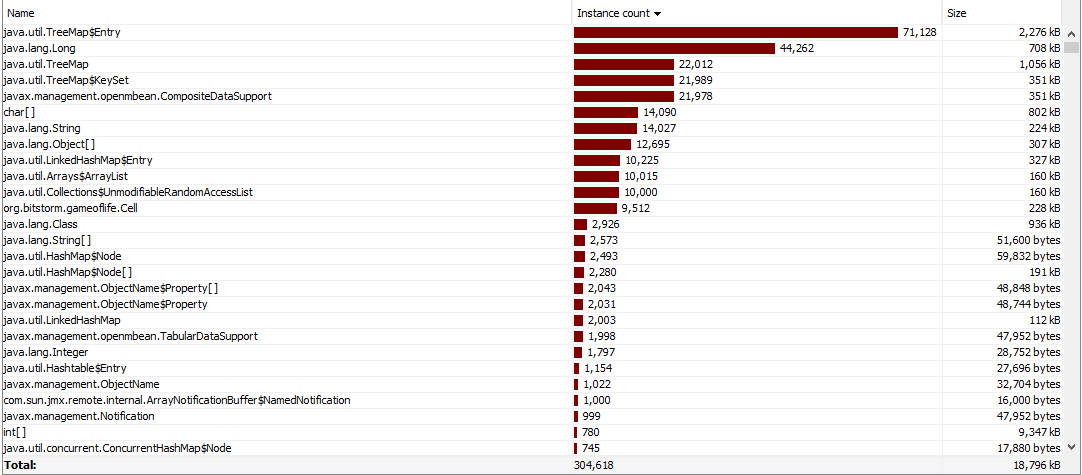
## Performance Optimalisatie

### C:\Users\Muhammed\Documents\School 2017-2018\ProgExpertProject\Knipsel.PNGMemory

De applicatie gebruikt ongeveer 26mb RAM, hoe meer cellen je hebt hoe meer RAM de applicatie gebruikt. Deze stijging komt voor omdat elk Cell een object op de heap is.   
De gebruikte Mb RAM blijft constant en dit is een gevolg van de Garbage Collection. Garbage collection gaat willekeurig door de heap en verwijdert ongebruikte objecten, hierdoor zal RAM geheugen vrijkomen. Indien er geen Garbage Collection is, zal de memory blijven stijgen tot er een Out of Memory Exception opgegooid wordt.  
Memory blijft constant dus er is ook geen sprake van memory leak in de applicatie.  
  
De Eden space gaat nooit tot 100% want er zijn niet veel objecten die aangemaakt worden en de garbage collection werkt om de zoveel seconden. In de Survivor space zitten er zeer weinig objecten en de Tenured Gen blijft zeer laag en constant.

De memory is goed genoeg geoptimaliseerd en we hoeven hier niks aan te passen.

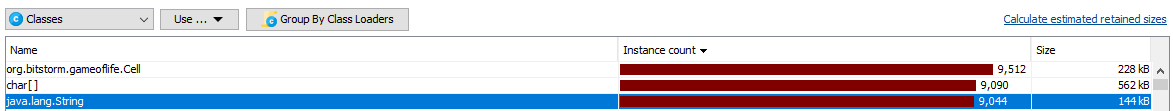
### Heap

We gaan dieper in de Heap kijken en zien welk objecten veel worden aangemaakt.

We zien dat er veel objecten zijn van Treemap, Char, String , Long. Omdat er veel instanties zijn, betekent het niet dat de size ook groot zal zijn. Als we kijken naar int[] heeft maar 780 instanties maar de grootte is wel 9347 kB.

Bij het appenden bij een String wordt er bij elk append een nieuw String object aangemaakt om dit te vermijden kan je gebruik maken van een StringBuilder. Dit zal zorgen dat er minder objecten van String worden aangemaakt.

We zien dat er een vermindering is van het aantal String objecten. Dit zal niet zo een groot performantie verschil hebben op ons applicatie omdat de aantal objecten weinig zijn. Maar bij een applicatie waar er meer dan 100.000 appends zijn zal de vermindering in objecten een groot performantie boost geven. En we zullen ook minder memory gebruiken aangezien we minder String objecten creëren.



### Final Keyword

Het final keyword definieert een constante binnen de Java syntax. Dit zorgt voor een snellere compileertijd. Performance winst door gebruik te maken van final is minimaal op zijn best. We spreken hier dan over microseconden.

Als voorbeeld nemen we de col en row die in de broncode gedefinieerd zijn als een short. Door deze als final te definiëren versnellen we het compileren. We kunnen ook potentiele memory leaks vermijden omdat er geen referenties kunnen zijn naar de variabele.

**Voorbeeld oplossing**

Originele code:

C:\Users\Muhammed\AppData\Local\Microsoft\Windows\INetCache\Content.Word\zonder Final.png

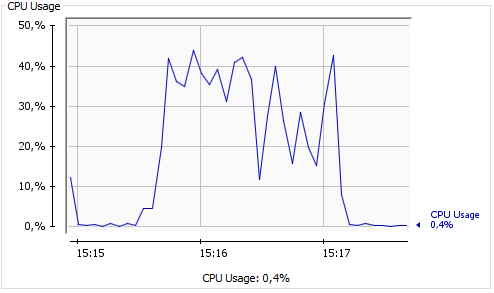
Onze verbetering:

C:\Users\Muhammed\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Final keyword.png

***Conclusie***

Verschil is bijna onmeetbaar aangezien dankzij de JIT compiler de functie verbeterd wordt per iteratie. Als we gebruik zouden maken van miljoenen cellen die op de zelfde moment actief zijn zouden we deze verschil wel kunnen meten. Maar in de theorie en aan de hand van voorbeelden kunnen we besluiten dat het de applicatie verbeterd.

### Multithreading



Het spel zelf draait niet op de main thread van de applicatie. Dit kan eenvoudig aangetoond worden door het venster snel te verkleinen en te vergroten terwijl het spel verder getekend moet worden.

De applicatie maakt gebruik van maar alleen één thread, namelijk “gameThread”. In deze thread zit alle logica van het spel, het is eigenlijk de main thread van het spel. Opdat maar alleen één thread voor het gehele spel wordt gebruikt, kan de CPU soms overbelast worden, uiteraard bij het veranderen van de snelheid van het spel naar ‘very fast’. Hierdoor lijkt het rendabel om threads bij te voegen die ervoor moeten zorgen dat de cores op een computer processor evenveel gebruikt worden.

Zoals u op de grafiek kunt zien, ontstaat de piek van de CPU usage wanneer het spel wordt versneld. Hierdoor moet de CPU de berekeningen van de cellen voor de volgende stap 10 keer sneller doen dan normaal. Na het wijzigen van de speelsnelheid naar ‘slow’, ziet u een onmiddellijke daling.

# Persoonlijke Reflectie

### Emre Demirsatan

### Muhammed Genc

# Bronvermelding

<https://nl.wikipedia.org/wiki/Game_of_Life>

<https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>

<https://stackoverflow.com/questions/40485/optimizing-conways-game-of-life>

<https://stackoverflow.com/questions/4279420/does-use-of-final-keyword-in-java-improve-the-performance>